THREAD SCHEDULING ON MULTIPROCESSOR SYSTEMS

BACKGROUND

1. FIELD

This disclosure relates generally to multi-thread applications on multi-processor systems, and, more specifically but not exclusively, to thread scheduling on a multi-processor system.

2. DESCRIPTION

A threaded application usually has data shared among its threads when running on symmetric multiprocessors ("SMP") and/or chip multiprocessors ("CMP"). The data sharing among different threads may be achieved in different ways but frequently done through a shared system-level memory. In a typical memory hierarchy in a multiprocessor system, a shared system-level memory between different processing cores has longer access latency for a processing core than a local cache of the processing core. Additionally, traffic (including coherency traffic) among different processing cores generated by excessive access to a shared system-level memory may saturate the bandwidth of the system interconnect (e.g., bus, ring, mesh, etc.). Therefore, it is desirable to investigate the data sharing behavior among different threads and reduce the cost of data transfer among threads.

BRIEF DESCRIPTION OF THE DRAWINGS

The features and advantages of the disclosed subject matter will become apparent from the following detailed description of the subject matter in which:

Figure 1 is a block diagram of an example multiprocessor system that uses a data sharing aware thread scheduling module, according to the disclosed subject matter in the present application;

Figure 2 is a block diagram of another example multiprocessor system that uses a data sharing aware thread scheduling module, according to the disclosed subject matter in the present application;

Figure 3 shows a block diagram of a data sharing aware thread scheduling module, according to the disclosed subject matter in the present application;

Figure 4 is a flowchart illustrating an example process for scheduling threads to target processors using information of data sharing behavior among different threads, according to the disclosed subject matter in the present application;

Figure 5 illustrates an example program that can be multithreaded in a multiprocessor system;

Figure 6 illustrates a code corresponding to the program shown in Figure 5, which is multithreaded for a multiprocessor system;

Figure 7 illustrates an example assembly code of the multi-threaded program shown in Figure 6;

Figure 8 illustrates an example assembly code of the multi-threaded program shown in Figure 6, which uses a data sharing aware thread scheduling method, according to the disclosed subject matter in the present application;

Figure 9 illustrates an example code of a multi-threaded video mining program, according to the disclosed subject matter in the present application; and

Figure 10 illustrates performance improvement of a multithread application running on a multiprocessor system, obtained by using a data sharing aware thread scheduling module, according to the disclosed subject matter in the present application.

DETAILED DESCRIPTION

According to embodiments of the subject matter disclosed in this application, a compiler in a multiprocessor system may compile a received multithreaded application, analyze data sharing behavior among multiple threads of the multithreaded application, and provide such information to a thread scheduler in the multiprocessor system. At run time, the thread scheduler may group together threads that share data frequently based on the data sharing information provided by the compiler, and schedule threads in the same group to processors in the same cluster. Processors in the same cluster have a shared storage device and have shorter access latency to the shared storage device than to the system-level memory. If there are not enough available processors in the same cluster, the rest of the threads may

be assigned to processors that are electronically in proximity to the cluster. Additionally, a feedback module may collect information on data sharing behavior among threads during run time and feedback such information to the thread scheduler. The thread scheduler may use the feedback information to regroup and reschedule the threads to processors at the next available scheduling time.

Reference in the specification to "one embodiment" or "an embodiment" of the disclosed subject matter means that a particular feature, structure or characteristic described in connection with the embodiment is included in at least one embodiment of the disclosed subject matter. Thus, the appearances of the phrase "in one embodiment" appearing in various places throughout the specification are not necessarily all referring to the same embodiment.

Figure 1 is a block diagram of an example multiprocessor system 100 that uses a data sharing aware thread scheduling module. System 100 may comprise multiple processors such as processor A (132A). Processors in system 100 may be connected to each other using a system interconnect 110. System interconnect 110 may be a Front Side Bus (FSB). Each processor may be connected to Input/Output (IO) devices as well as system-level memory 120 through the system interconnect. The system-level memory may be a Dynamic Random Access Memory (DRAM), a Synchronous DRAM (SDRAM), a Double Data Rate (DDR) SDRAM, or other types of memory devices. Each processor (e.g., processor A (132A)) may have its own cache with one or more levels (e.g., level 1 (L1) and level 2 (L2) caches, not shown

in the figure). In one embodiment, several processors (e.g., processor A (132A) through processor M (132M)) may have a shared local-level storage device such as cache 138. Cache 138 may be connected with processor 132A through processor 132M via a connection mechanism 136. Connection mechanism 136 may be a local interconnect (e.g., local bus), a connection ring, or a cross-bar.

Processors 132A through 132M, connection mechanism 136, and cache 138, together may form a cluster (cluster 1 (130)). Cluster 1 may be connected with other clusters (e.g., cluster N (140)), individual processors (not shown in the figure), IO devices and system-level memory 130, and other devices via system interconnect 110. In one embodiment, an individual processor (not shown in the figure) may have multiple processing cores. Each processing core may have its own cache and all of the processing cores together may have shared local cache. Such a processor with multiple processing cores may be treated similarly as a cluster (e.g., cluster 1). For the convenience of description, a processing core inside a processor may be treated in the same way as a single-core processor. Thus, the word "processor" may be used to represent either a single-core processor or a processing core in a multi-core processor.

A data sharing aware thread scheduling module (e.g., 134A, 134M) may be included in a processor (such as processor 132A or 132M). The thread scheduling module may comprise a compiler, a thread scheduler, and a feedback module. The compiler may receive a multithreaded application and compile the application into one or more object code which is specific to

the underlying architecture. The thread scheduler may be a part of, or associate with, an operating system (OS). The thread scheduler schedules multiple threads of the multithreaded application to different processors. The feedback module may observe actual data sharing behavior among threads during execution and provide the actual data sharing information to the thread scheduler, which will use the feedback information to fine tune its thread scheduling next time.

When two or more threads share data frequently, such data sharing is often achieved through shared system-level memory (e.g., memory 120). Access to a system-level memory has longer latency by a processor (e.g., processor 132A) than access to a local-level memory (e.g., cache 138) by the processor. Additionally, frequent access to a system-level memory consumes quite a bit bandwidth of the system interconnect (e.g., 110) and may even cause congestion on the system interconnect. Furthermore, coherence traffic generated by access to a system-level memory typically has high overhead.

A conventional thread scheduler does not have much knowledge on data sharing behavior among multiple threads, and when it schedules multiple threads, it mainly focuses on memory hierarchy optimization. According to one embodiment of the subject matter in the present application, the thread scheduler may utilize information on data sharing behavior among threads provided by the compiler, which is able to identify such information when compiling a multithreaded application. Additionally, the thread scheduler has the ability of knowing the underlying architecture of the system, for example, which processors are in the same cluster and which clusters are in proximity

to each other electronically. Clusters that are in proximity to each other electronically include such clusters where data sharing among their processors requires less time, causes less traffic on the system interconnect, and/or incurs less coherence overhead, than data sharing among processors from other clusters. Such clusters may include those that are close to each other in topology. With data sharing information from the compiler plus knowledge of underlying architectural characteristics, the thread scheduler (e.g., 134A) may schedule those tightly coupled threads (with frequent data sharing among them) to processors in the same cluster. If there are not enough available processors in any cluster, the thread scheduler may schedule those tightly coupled threads to processors in those clusters that are in proximity to each other electronically.

Figure 2 is a block diagram of another example multiprocessor system 200 that uses a data sharing aware thread scheduling module. In system 200, system interconnect 210 that connects multiple clusters (e.g., 220A, 220B, 220C, and 220D) is a links-based point-to-point connection. Each cluster may connect to the system interconnect through a links hub (e.g., 230A, 230B, 230C, and 230D). In some embodiments, a links hub may be co-located with a memory controller, which coordinates traffic to/from a system memory (not shown in the figure). Similar to cluster 1 (130) shown in Figure 1, clusters 220A, 220B, 220C, and 220D may comprise two or more processors (or processing cores), which are connected with a shared local storage device (e.g., last level cache). Each processor may have its own local caches.

A processor in a cluster may have a data sharing aware thread scheduling module which may comprise a compiler, a thread scheduler, and a feedback module. The compiler receives a multithreaded application and compiles it into object code specific to the underlying hardware. The thread scheduler may schedule multiple threads of the multithreaded application to various processors. The feedback module may provide actual data sharing information among threads during execution to the thread scheduler, which will use such information for fine tuning thread scheduling next time.

According to one embodiment of the subject matter disclosed in the present application, the compiler may provide the thread scheduler data sharing information among threads. The thread scheduler may also obtain the underlying hardware configuration from the OS or a hardware abstract layer (e.g., virtual machine). Based on the data sharing information and the hardware structural information, the thread scheduler may assign tightly coupled threads to those processors which are in proximity to each other electronically, for example, processors in the same cluster, or processors in clusters that are in proximity to each other electronically if there are not enough available processors in one cluster.

Figure 3 shows a block diagram of a data sharing aware thread scheduling module 300 according to the subject matter disclosed in the present application. The module may comprise a compiler 320, a thread scheduler 330, and a feedback module 350. For the purpose of convenient description, the figure also includes a multithreaded application 310 and target processors 340. Compiler 320 may receive a multithreaded application and

compile the application into one or more object code which is specific to the underlying architecture. Additionally, the compiler analyzes data sharing behavior among different threads and obtains data sharing information before the threads are executed on target processors. The compiler then provides such data sharing information to the thread scheduler.

Thread scheduler 330 may be a part of, or associate with, an operating system (OS). Based on the data sharing information provided by the compiler, the thread scheduler may divide all of the threads into different groups. Threads within the same group share data with each other frequently. In one embodiment, the compiler may group all the threads into different groups based on their data sharing behavior obtained during compilation and pass the grouping information to the thread scheduler. Additionally, the thread scheduler is capable of obtaining structural characteristics of the underlying architecture of a multiprocessor system such as, for example, which processors are in the same cluster and which clusters are in proximity to each other electronically. If the OS has the knowledge of hardware structural features, the thread scheduler may obtain such hardware structural information from the OS. If the OS does not have direct knowledge of hardware structural features (e.g., in a situation where a virtual machine exists between the OS and the underlying hardware), the OS or the thread scheduler may still get such hardware information by invoking certain application program interfaces (API's).

With the hardware information, the thread scheduler may assign threads in the same group to processors in the same cluster. If there are not

enough processors available in any cluster, the thread schedule may identify a cluster that has the highest number of available processors, assign a corresponding number of threads in a group to processors in the cluster, and assign the rest of threads in the group to processors in another cluster that is in proximity to the cluster electronically. In case the cluster that has the highest number of available processors cannot host all of the threads in a group and does not have any cluster in proximity electronically that has enough available processors that can host the rest of the threads in the group, the thread scheduler may assign threads in the group to clusters that are in proximity to each other electronically and that together have enough processors available to host all of the threads in the group.

Feedback module 350 may observe actual data sharing behavior among threads during execution and provide this actual data sharing information to the thread scheduler. The thread scheduler may use such feedback information on actual data sharing behavior to regroup threads of the multithreaded application and reschedule the threads to target processors according to the regrouping result, at the next available thread scheduling time. The thread scheduler may first examine whether the feedback information supports regrouping and rescheduling. For example, if the actual data sharing behavior during execution does not deviate from the current grouping significantly, the thread scheduler may decide not to regroup or reschedule the threads. In one embodiment, the thread scheduler may decide whether to regroup and reschedule threads of the application based on the execution status of the application. For example, if the application is close

to completion when the feedback information regarding actual data sharing behavior is received and supports regrouping and rescheduling, the thread scheduler may choose not to regroup or reschedule threads at the next scheduling time.

Figure 4 is a flowchart illustrating an example process 400 for scheduling threads to target processors using information of data sharing behavior among different threads, according to the subject matter disclosed in the present application. At block 410, a multithreaded application may be received and passed to a compiler. At block 420, the multithreaded application may be compiled by the compiler. While compiling the application, the compiler may analyze data sharing behavior among threads of the application and collect data sharing information at block 430. Also at block 430, the data sharing information may be provided to a thread scheduler by the compiler. At block 440, threads that share data frequently, in other words, threads that are tightly coupled, may be identified based on the data sharing information provided by the compiler. All of the threads of the application may be placed into different groups, with those tightly coupled threads being in the same group. In one embodiment, the compiler may group all of the threads of the application using the data sharing information obtained during compilation and then pass the grouping information to the thread scheduler.

At block 450, threads in each group may be scheduled to processors that are in proximity to each other electronically. If there is any cluster that has enough available processors, threads in a group will be scheduled to processors in the same cluster; otherwise, threads will be scheduled to

processors in the minimum number of clusters that are in proximity to processors in the cluster. At block 460, threads that have been assigned may be executed on target processors. At block 470, a feedback module may observe actual data sharing behavior during execution on the target processors. At block 480 such actual data sharing information may be provided to the thread scheduler. Upon receiving such information, the thread scheduler will decide whether rescheduling of threads is necessary at the next scheduling time. If it is, the thread scheduler may regroup threads or fine tune previous grouping and reschedule the threads based on regrouping or fine tuning at the next available time.

Figure 5 illustrates an example program 500 that can be multithreaded in a multiprocessor system. This is a simple pointer chasing application. Depending on the size of the list p, it may take a long time to complete execution of this simple application. Thus, it may be necessary to multithread this application and run multiple threads in parallel. There are several ways to achieve parallel implementation of an application in multiple threads. One way is to use OpenMP, a parallel programming language, which has been supported by many compilers, such as Intel® C++ and Fortran compilers. The OpenMP programming model provides a work-queue execution model with task-queue and task primitives to allow users to efficiently exploit parallelism among irregular patterns of dynamic data structures and/or those with complicated control structures such as recursion. In the following description, the OpenMP programming model is used as an example to

illustrate how a data sharing aware thread scheduling scheme, as disclosed in this application, works.

Figure 6 illustrates a code 600 corresponding to the program shown in Figure 5, which is multithreaded for a multiprocessor system using the OpenMP programming model. The main differences between code 600 and code 500 are two added lines (Line 615 and line 635) in code 600, which parallelize the pointer chasing application shown in Figure 5.

Figure 7 illustrates an example assembly code 700 of the multi-threaded program shown in Figure 6. Assembly code 700 is obtained by compiling code 600. Lines 750 through 770 are assembly translations for the master thread. Code 700 illustrates how thread scheduling is done without considering data sharing behavior among threads. Line 710 creates a team of threads. The compiler may use data sharing information among threads obtained during compilation to create a team of threads by placing those tightly coupled threads into the same team. However, assembly 700 does not provide a thread scheduler such data sharing information among threads. Thus, when the thread scheduler schedules a team of threads, it does not necessarily assign threads in the same team to processors in the same cluster. In other words, even if the data sharing information is considered in creating a team of threads, without informing the thread scheduler of such consideration, the resulting scheduling will not be based on the data sharing information obtained by the compiler.

Figure 8 illustrates an example assembly code 800 of the multi-threaded program shown in Figure 6, which uses a data sharing aware thread

scheduling method, according to the subject matter disclosed in the present application. The main difference between code 800 and code 700 is line 815 which is not in code 700. In code 800, line 810 creates a team of threads with consideration of data sharing information among threads obtained by the compiler. Line 815 then informs the thread scheduler that threads in the same team should be scheduled to processors in the same cluster or in clusters that are in proximity to each other electronically.

Figure 9 illustrates an example code 900 of a multi-threaded video mining program, according to the subject matter disclosed in the present application. Code 900 is a sample from a video mining application. This application uses a task-queue to build a pipeline based parallel version. Within one task-queue, one thread is responsible for video decoding, and other worker threads perform feature extraction from decoded frames. In this application, the number of threads is obtained by invoking the function omp_get_num_threads(). Following the disclosed data sharing aware scheduling scheme, the compiler can easily add the scheduling hint information (e.g., line 815 in Figure 8) and the thread scheduler can then utilize this information to achieve efficient thread scheduling.

Figure 10 illustrates a performance improvement of a multithread application running on a multiprocessor system, obtained by using a data sharing aware thread scheduling module, according to the disclosed subject matter in the present application. An experiment was conducted by using a 32-way shared memory multiprocessor system to study the effectiveness of the disclosed data sharing aware thread scheduling scheme. Each processor

is equipped with an 8KB L1 cache, 512KB L2 cache, and 4M L3 Cache. Each cluster has 4 processors that share a 32MB L4 cache. The workload used for the study is a video mining application and a hybrid parallel scheme is employed, which exploits both data and task level parallelism to parallelize the application. With the hint information of data sharing provided by the compiler, the thread scheduler allocates threads in a team to processors in the same cluster (in this particular experiment, a team of threads has 4 threads).

For comparison purposes, two different cases are studied. In case 2, the disclosed data sharing thread scheduling scheme is used. In case 1, closely coupled threads, among which there is a lot of data sharing, are intentionally allocated to processors scattered in different clusters. Performance of each case is compared with the default scheduling scheme, which does not allocate threads without considering data sharing information among the threads. As shown in Figure 10, the x-axis shows the number of processors which are actually used in the multiple processor system; the y-axis shows the performance ratio between each of the two studied cases and the default scheduling scheme. Experiment results clearly show that performance in case 1 is consistently below performance in the default case for different number of processors. However, performance in case 2, which uses the disclosed data sharing aware thread scheduling scheme, is significantly better than performance in the default case for any number of processors studied in the experiment.

Although an example embodiment of the disclosed subject matter is described with reference to block and flow diagrams in Figures 1-10, persons of ordinary skill in the art will readily appreciate that many other methods of implementing the disclosed subject matter may alternatively be used. For example, the order of execution of the blocks in flow diagrams may be changed, and/or some of the blocks in block/flow diagrams described may be changed, eliminated, or combined.

In the preceding description, various aspects of the disclosed subject matter have been described. For purposes of explanation, specific numbers, systems and configurations were set forth in order to provide a thorough understanding of the subject matter. However, it is apparent to one skilled in the art having the benefit of this disclosure that the subject matter may be practiced without the specific details. In other instances, well-known features, components, or modules were omitted, simplified, combined, or split in order not to obscure the disclosed subject matter.

Various embodiments of the disclosed subject matter may be implemented in hardware, firmware, software, or combination thereof, and may be described by reference to or in conjunction with program code, such as instructions, functions, procedures, data structures, logic, application programs, design representations or formats for simulation, emulation, and fabrication of a design, which when accessed by a machine results in the machine performing tasks, defining abstract data types or low-level hardware contexts, or producing a result.

For simulations, program code may represent hardware using a hardware description language or another functional description language which essentially provides a model of how designed hardware is expected to perform. Program code may be assembly or machine language, or data that may be compiled and/or interpreted. Furthermore, it is common in the art to speak of software, in one form or another as taking an action or causing a result. Such expressions are merely a shorthand way of stating execution of program code by a processing system which causes a processor to perform an action or produce a result.

Program code may be stored in, for example, volatile and/or non-volatile memory, such as storage devices and/or an associated machine readable or machine accessible medium including solid-state memory, hard-drives, floppy-disks, optical storage, tapes, flash memory, memory sticks, digital video disks, digital versatile discs (DVDs), etc., as well as more exotic mediums such as machine-accessible biological state preserving storage. A machine readable medium may include any mechanism for storing, transmitting, or receiving information in a form readable by a machine, and the medium may include a tangible medium through which electrical, optical, acoustical or other form of propagated signals or carrier wave encoding the program code may pass, such as antennas, optical fibers, communications interfaces, etc. Program code may be transmitted in the form of packets, serial data, parallel data, propagated signals, etc., and may be used in a compressed or encrypted format.

Program code may be implemented in programs executing on programmable machines such as mobile or stationary computers, personal digital assistants, set top boxes, cellular telephones and pagers, and other electronic devices, each including a processor, volatile and/or non-volatile memory readable by the processor, at least one input device and/or one or more output devices. Program code may be applied to the data entered using the input device to perform the described embodiments and to generate output information. The output information may be applied to one or more output devices. One of ordinary skill in the art may appreciate that embodiments of the disclosed subject matter can be practiced with various computer system configurations, including multiprocessor or multiple-core processor systems, minicomputers, mainframe computers, as well as pervasive or miniature computers or processors that may be embedded into virtually any device. Embodiments of the disclosed subject matter can also be practiced in distributed computing environments where tasks may be performed by remote processing devices that are linked through a communications network.

Although operations may be described as a sequential process, some of the operations may in fact be performed in parallel, concurrently, and/or in a distributed environment, and with program code stored locally and/or remotely for access by single or multi-processor machines. In addition, in some embodiments the order of operations may be rearranged without departing from the spirit of the disclosed subject matter. Program code may be used by or in conjunction with embedded controllers.

While the disclosed subject matter has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various modifications of the illustrative embodiments, as well as other embodiments of the subject matter, which are apparent to persons skilled in the art to which the disclosed subject matter pertains are deemed to lie within the scope of the disclosed subject matter.